CSE 332: Data Structures & Parallelism
Lecture 20: Shortest Paths

Arthur Liu

Summer 2022

Announcements

- End of quarter is coming up!!
  - Unfortunately, this also means there is not a lot of flexibility since grades are due
  - EX16 cancelled!
- Last day of OH is on Wed 10/17
- Hans will be giving the Friday lecture on MST!

Announcements

- Reminder your final is on two days, Section 10/18, Lecture 10/19
  - Make sure to be in your correct quiz section on Thursday for pt1. of the exam! We will take attendance, so bring student ID to section
- Final Review Session: MOR 220 Wed 10/17 from 3:00-4:00pm
- Exam Topics and Practice Exams on the website!
  - Make sure to look at some past finals to practice!

Outline for Today

- Dijkstra’s Algorithm

Shortest Path Applications

- Network routing
- Driving directions
- Cheap flight tickets
- Critical paths in project management (see textbook)
- …

Single source shortest paths

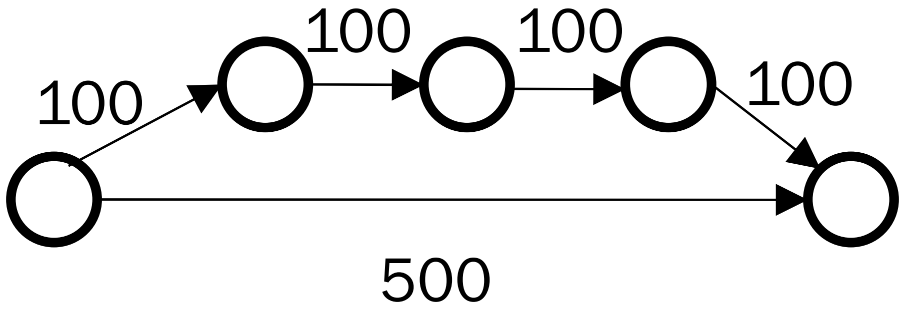
- Done: BFS to find the minimum path length from **s** to **t** in $O(|E|+|V|)$
- Actually, we found the minimum path length from **s** to *every node*
  - Still $O(|E|+(|V|)$
  - No faster way for a "distinguished" destination in the worst-case
- Now: Weighted graphs

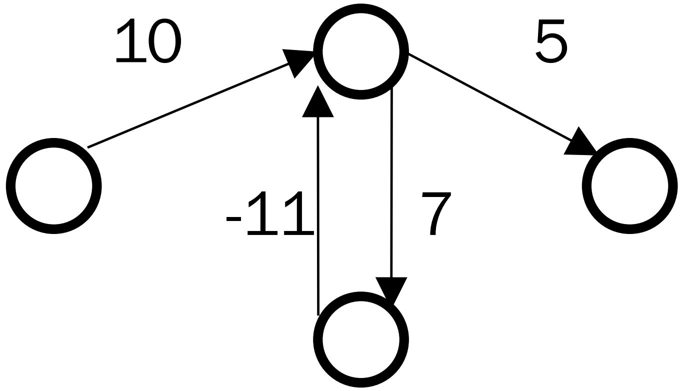
Given a weighted graph and node **s**,

find the minimum-cost path from **s** to every node

- As before, asymptotically no harder than for one destination
- Unlike before, BFS will not work

Not as easy





Why BFS won't work: Shortest path may not have the fewest edges

- Annoying when this happens with costs of flights

We will assume there are no negative weights

- *Problem* is *ill-defined* if there are negative-cost *cycles*
- *Today's algorithm* is *wrong* if *edges* can be negative

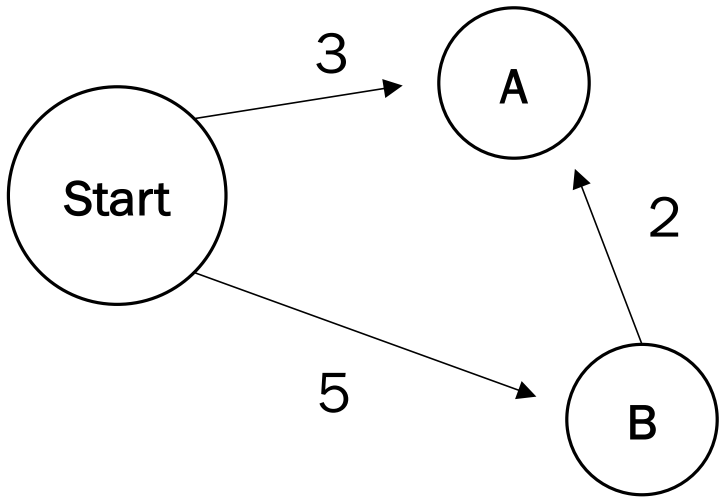
Dijkstra’s Algorithm

- Named after its inventor Edsger Dijkstra (1930-2002)
  - Truly one of the “founders” of computer science;
    1972 Turing Award; this is just one of his many contributions
  - Sample quotation: “computer science is no more about computers than astronomy is about telescopes”

- The idea: reminiscent of BFS, but adapted to handle weights
  - Grow the set of nodes whose shortest distance has been computed
  - Nodes not in the set will have a “best distance so far”
  - A priority queue will turn out to be useful for efficiency

Dijkstra’s Intuition

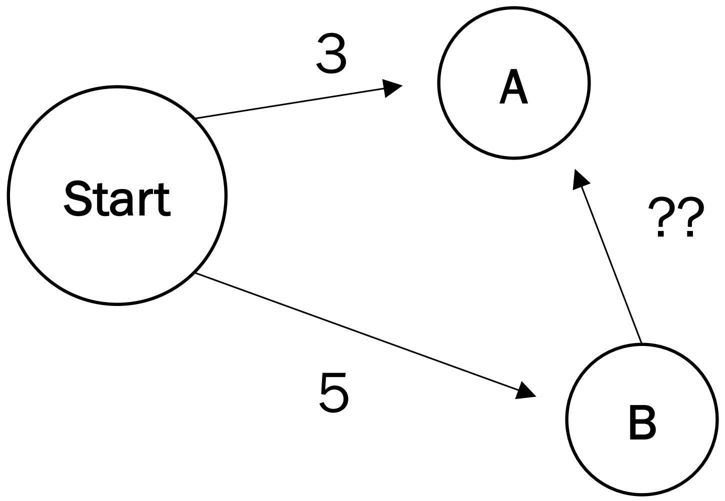
At each step, process the next closest vertex to our start.



Without trying other paths to A, why do we know that the shortest path to A must be a total of cost 3?

Dijkstra’s Intuition

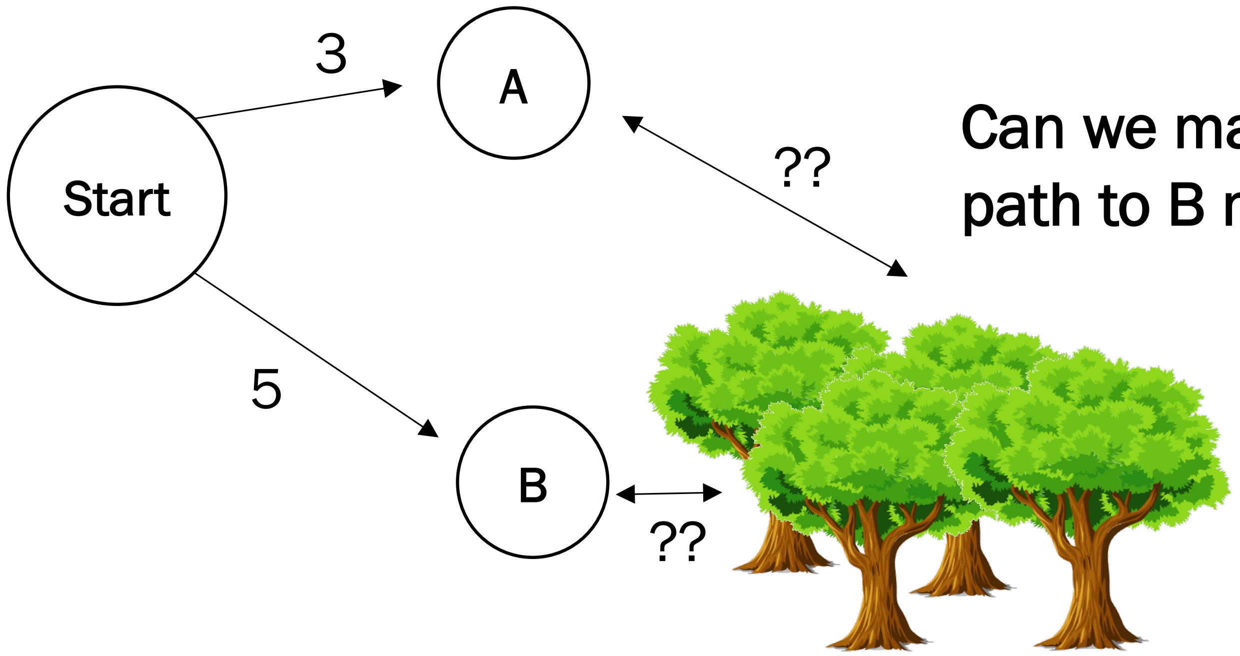
At each step, process the next closest vertex to our start.



Without trying other paths to A, why do we *still* know that the shortest path to A must be a total of cost 3?

Dijkstra’s Intuition

At each step, process the next closest vertex to our start.



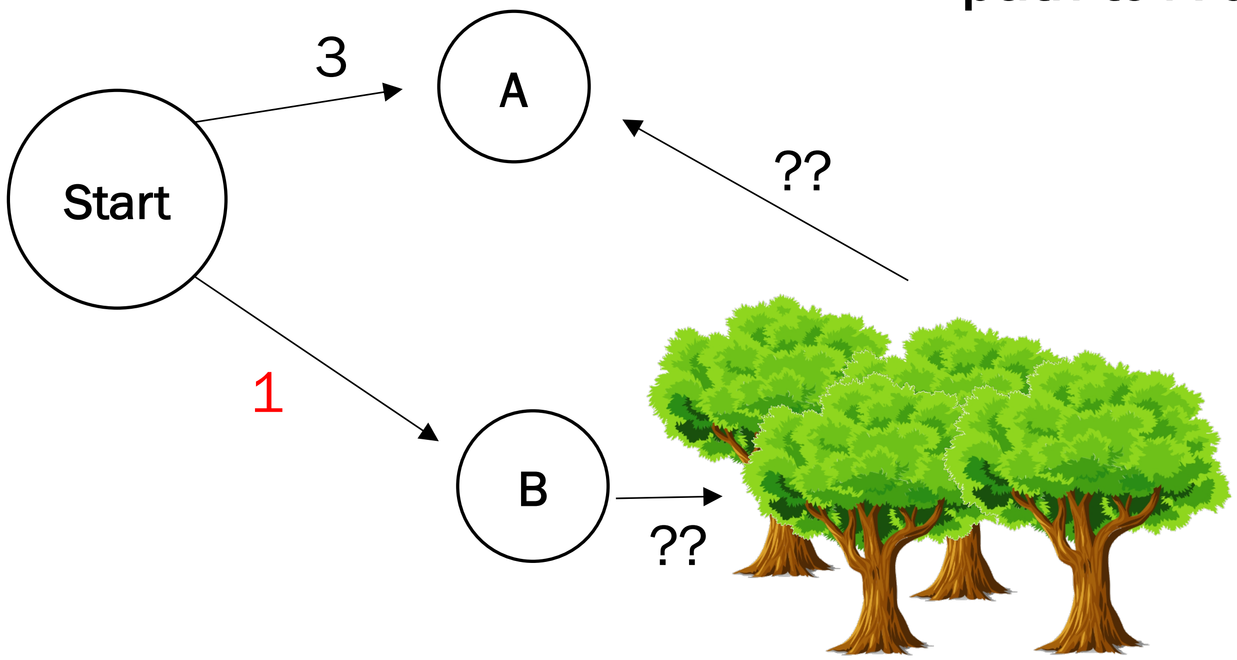
Without trying other paths to A, why do we *still, still* know that the shortest path to A must be a total of cost 3?

Can we make the claim that the shortest path to B must be 5?

Dijkstra’s Intuition

At each step, process the next closest vertex to our start.

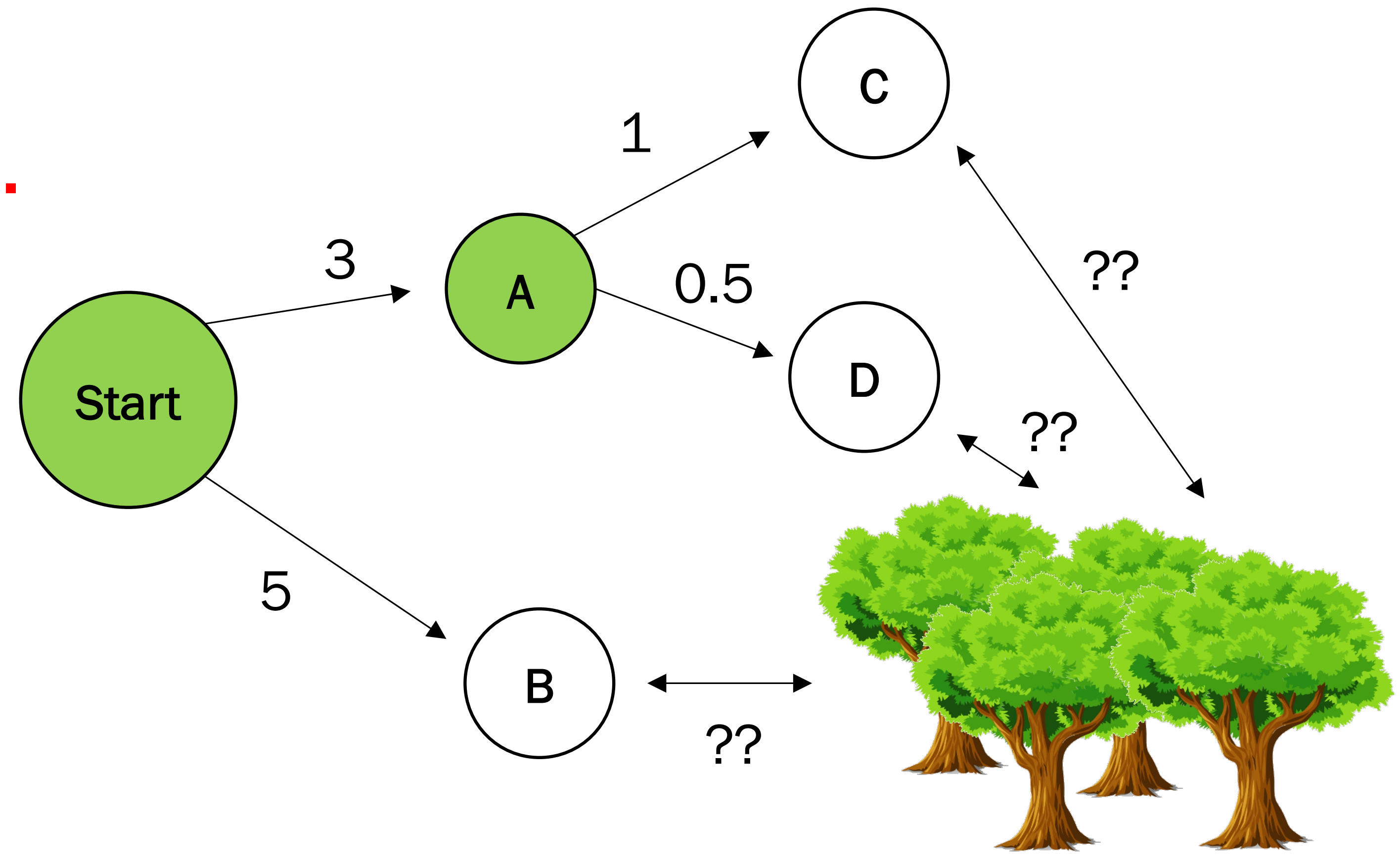
What if the weights were different? Do we still know that the shortest path to A costs 3?



Dijkstra’s Intuition

At each step, process the next closest vertex to our start, which we know must be the shortest possible distance to that node.

Rinse and repeat.

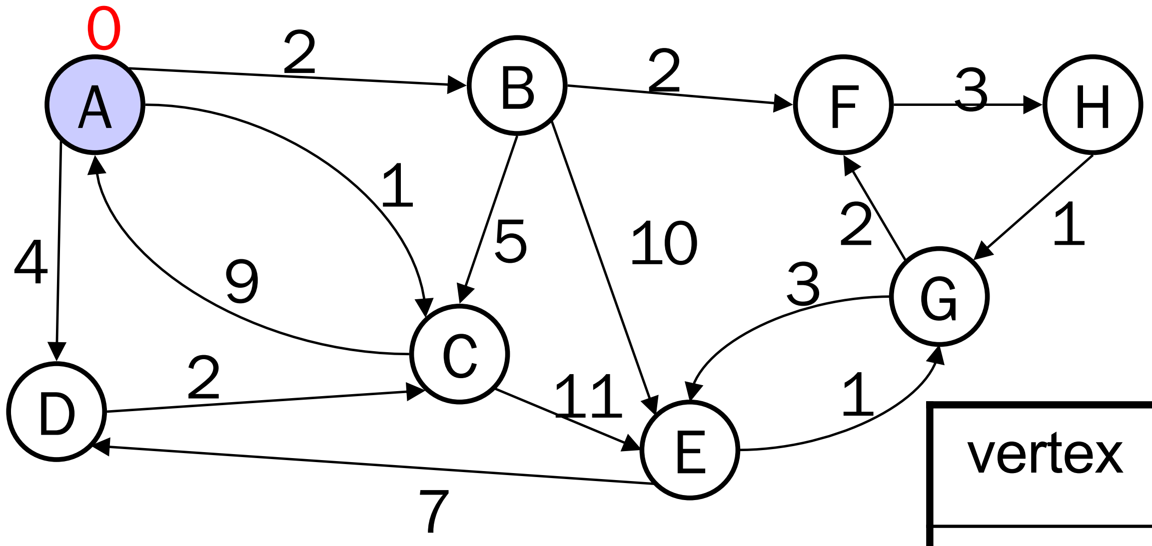


The Algorithm

1. For each node **`v`**, set **`v.cost = ∞`** and **`v.known = false`**
2. Set **`source.cost = 0`**
3. While there are unknown nodes in the graph
   a) Select the unknown node **`v`** with lowest cost
   b) Mark **`v`** as known
   c) For each edge **`(v,u)`** with weight **`w`**, if **`u`** is unknown,

```
c1 = v.cost + w // cost of best path through v to u
c2 = u.cost  // cost of best path to u previously known
if(c1 < c2){ // if the path through v is better
  u.cost = c1
  u.path = v // for computing actual paths
}
```

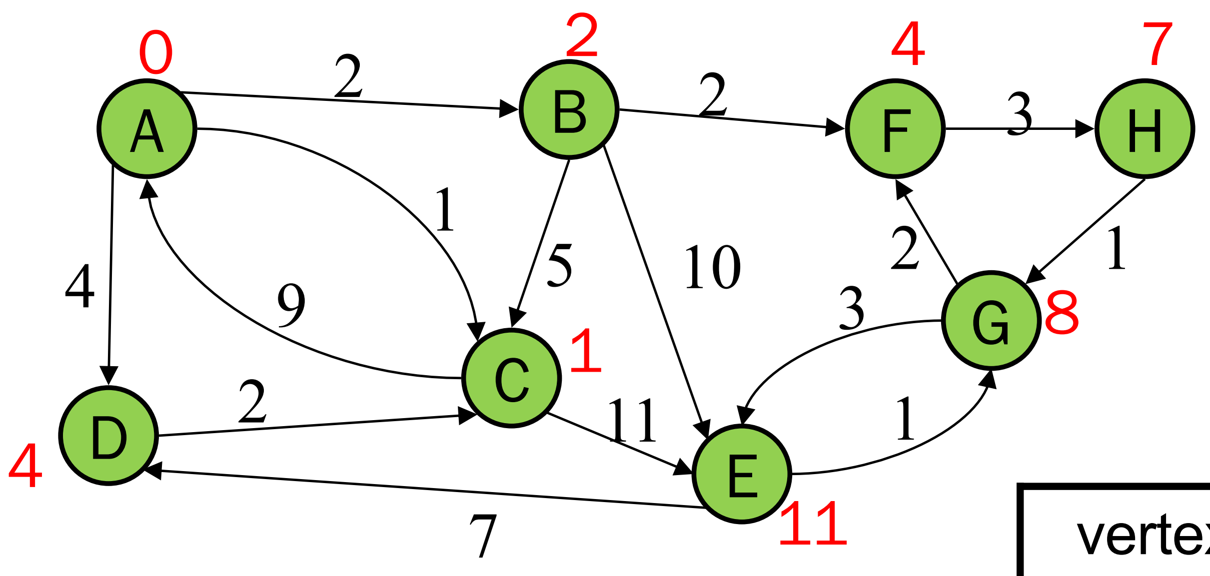
Example #1



Order Added to Known Set:

vertex	known?	cost	path
A			
B			
C			
D			
E			
F			
G			
H			

Example #1



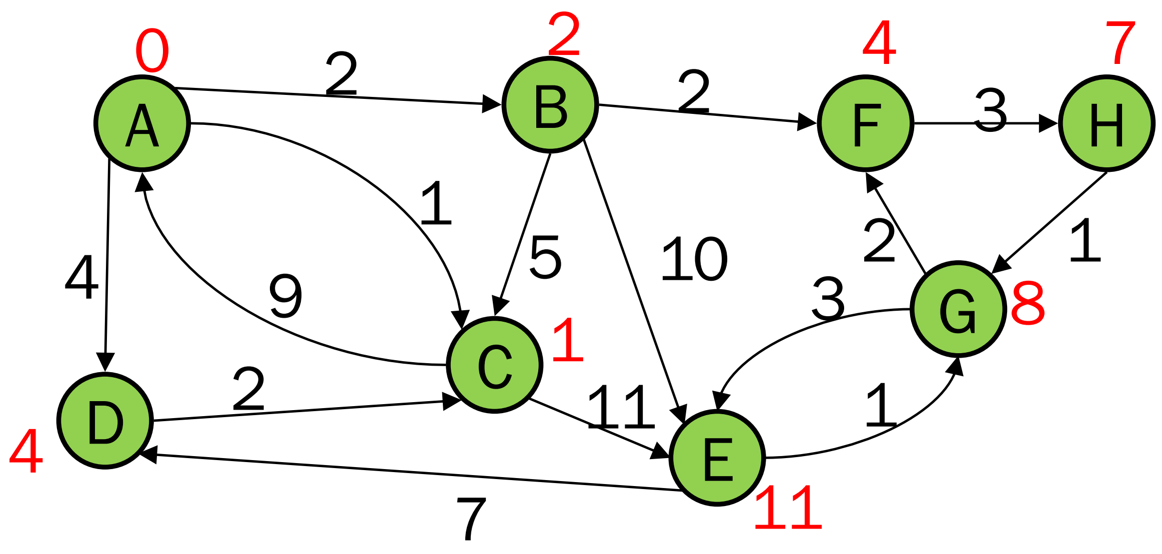
vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H, G, E

Interpreting the Results

- Now that we're done, how do we get the path from, say, A to E?



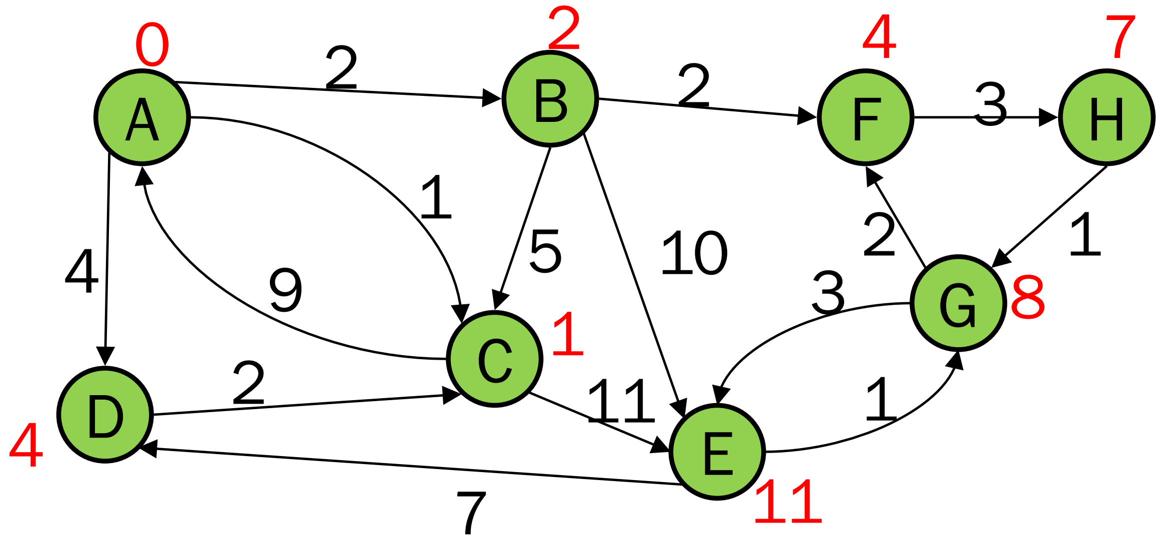
Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Stopping Short

- How would this have worked differently if we were only interested in:
  - The path from A to G?
  - The path from A to D?

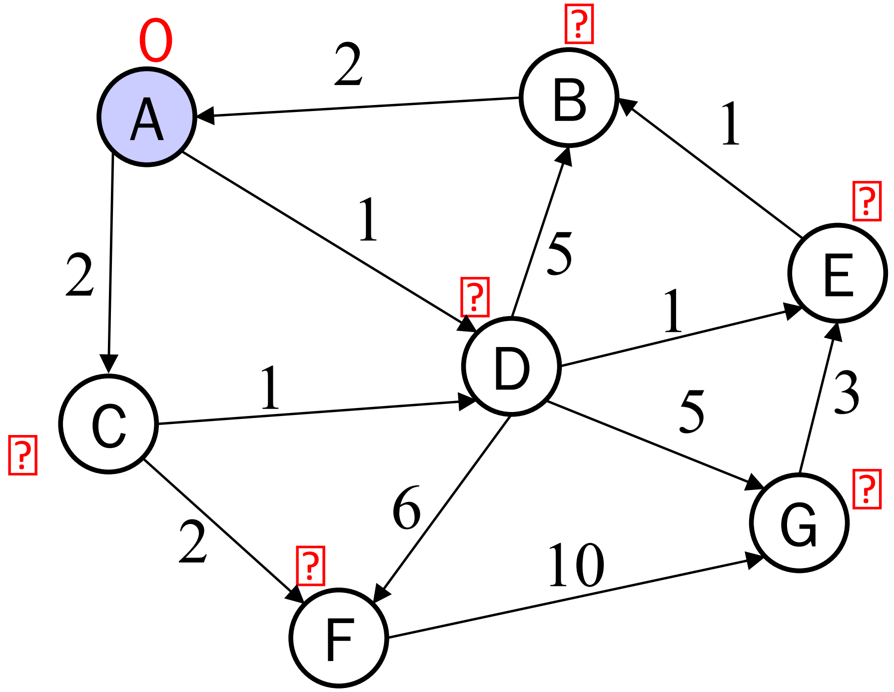


Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

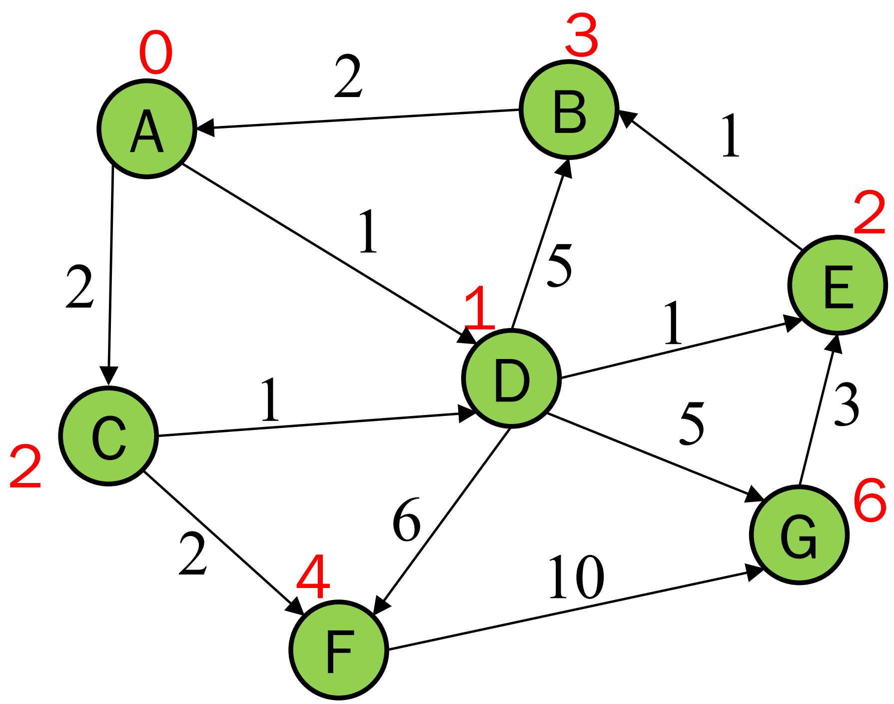
Example #2



Order Added to Known Set:

vertex	known?	cost	path
A		0	
B			
C			
D			
E			
F			
G			

Example #2



Order Added to Known Set:

A, D, C, E, B, F, G

vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G	Y	6	D

Features

- When a vertex is marked known,
  the cost of the shortest path to that node is known
  - The path is also known by following back-pointers

- While a vertex is still not known,
  another shorter path to it might still be found
  - The current cost we have is an upper-bound though!

Note: The “Order Added to Known Set” is not important

- A detail about how the algorithm works (client doesn’t care)
- Not used by the algorithm (implementation doesn’t care)
- It is sorted by path-cost, resolving ties in some way

A Greedy Algorithm

- Dijkstra's algorithm
  - For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges

- An example of a *greedy algorithm*:
  - At each step, irrevocably does what seems best at that step
    - A locally optimal step, not necessarily globally optimal
  - Once a vertex is known, it is not revisited
    - Turns out to be globally optimal

When greed fails us

Making change – use fewest # of coins possible for 15¢

25, 10, 5, 1

25, 12, 10, 5, 1

Where are we?

- What should we do after learning an algorithm?
  - Prove it is correct
    - Not obvious!
    - We will sketch the key ideas
  - Analyze its efficiency
    - Will do better by using a data structure we learned earlier!

Correctness: Intuition

Rough intuition:

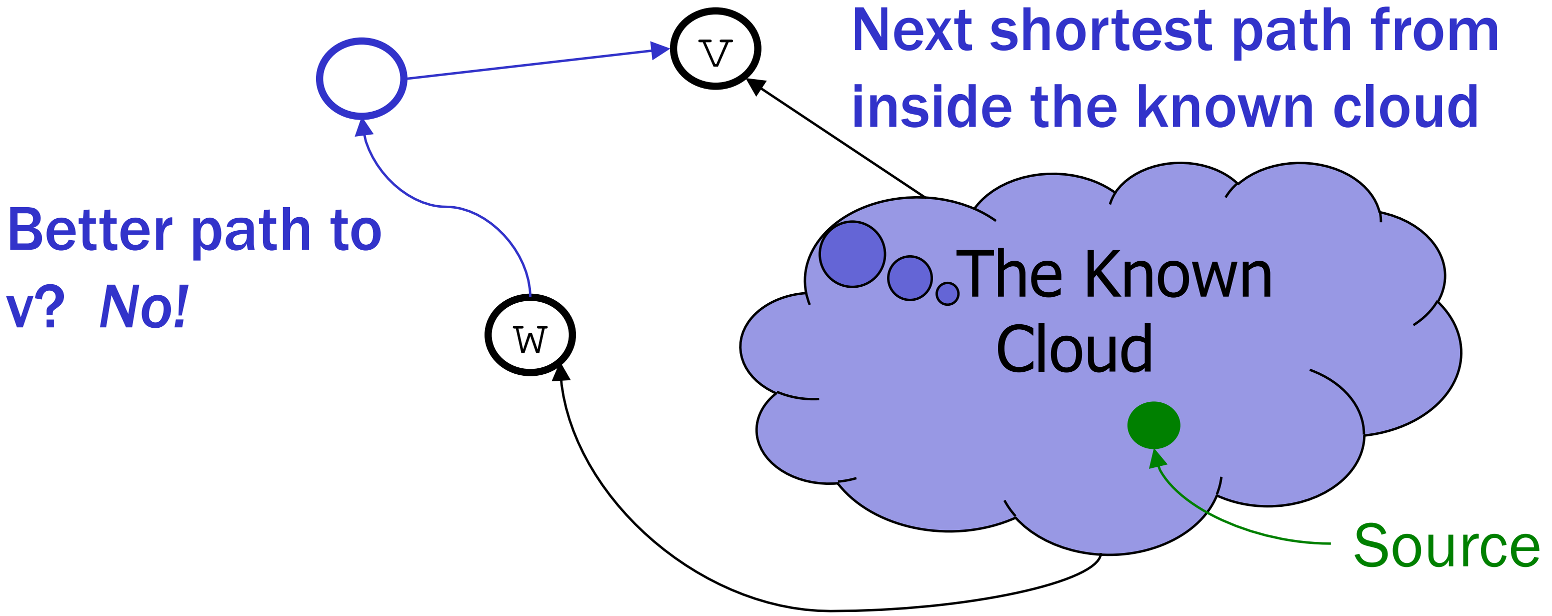
All the “known” vertices have the correct shortest path

- True initially: shortest path to start node has cost 0
- If it stays true every time we mark a node “known”, then by induction this holds and eventually everything is “known”

Key fact we need: When we mark a vertex “known” we won’t discover a shorter path later!

- This holds only because Dijkstra’s algorithm picks the node with the next shortest path-so-far
- The proof is by contradiction…

Correctness: The Cloud (Rough Idea)



Suppose v is the next node to be marked known ("added to the cloud")

- The best-known path to v must have only nodes "in the cloud"
  - Since we've selected it, and we only know about paths through the cloud to a node right outside the cloud
- Assume (for contradiction) the actual shortest path to v is different
  - It won't use only cloud nodes, (or we would know about it), so it must use non-cloud nodes
  - Let w be the *first* non-cloud node on this path.
  - The part of the path up to w is already known and must be shorter than the best-known path to v. So v would not have been picked.

Contradiction!

Efficiency, first approach

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
       if(b.cost + weight((b,a)) < a.cost){
         a.cost = b.cost + weight((b,a))
         a.path = b
       }
}
```

Improving asymptotic running time

- So far: $O(|V|^2 + |E|)$

- We had a similar "problem" with topological sort being $O(|V|^2 + |E|)$ due to each iteration looking for the node to process next
  - We solved it with a queue of zero-degree nodes
  - But here we need the lowest-cost node and costs can change as we process edges

- Solution?

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
      if(b.cost + weight((b,a)) < a.cost){
        decreaseKey(a,"new cost – old cost")
        a.path = b
      }
}
```

Dense vs. sparse again

First approach: $O(|V|^2 + |E|)$

Second approach: $O(|V|\log|V| + |E|\log|V|)$

So which is better?

Dense vs. sparse again

First approach: $O(|V|^2 + |E|)$ or: $O(|V|^2)$

Second approach: $O(|V|\log|V|+|E|\log|V|)$

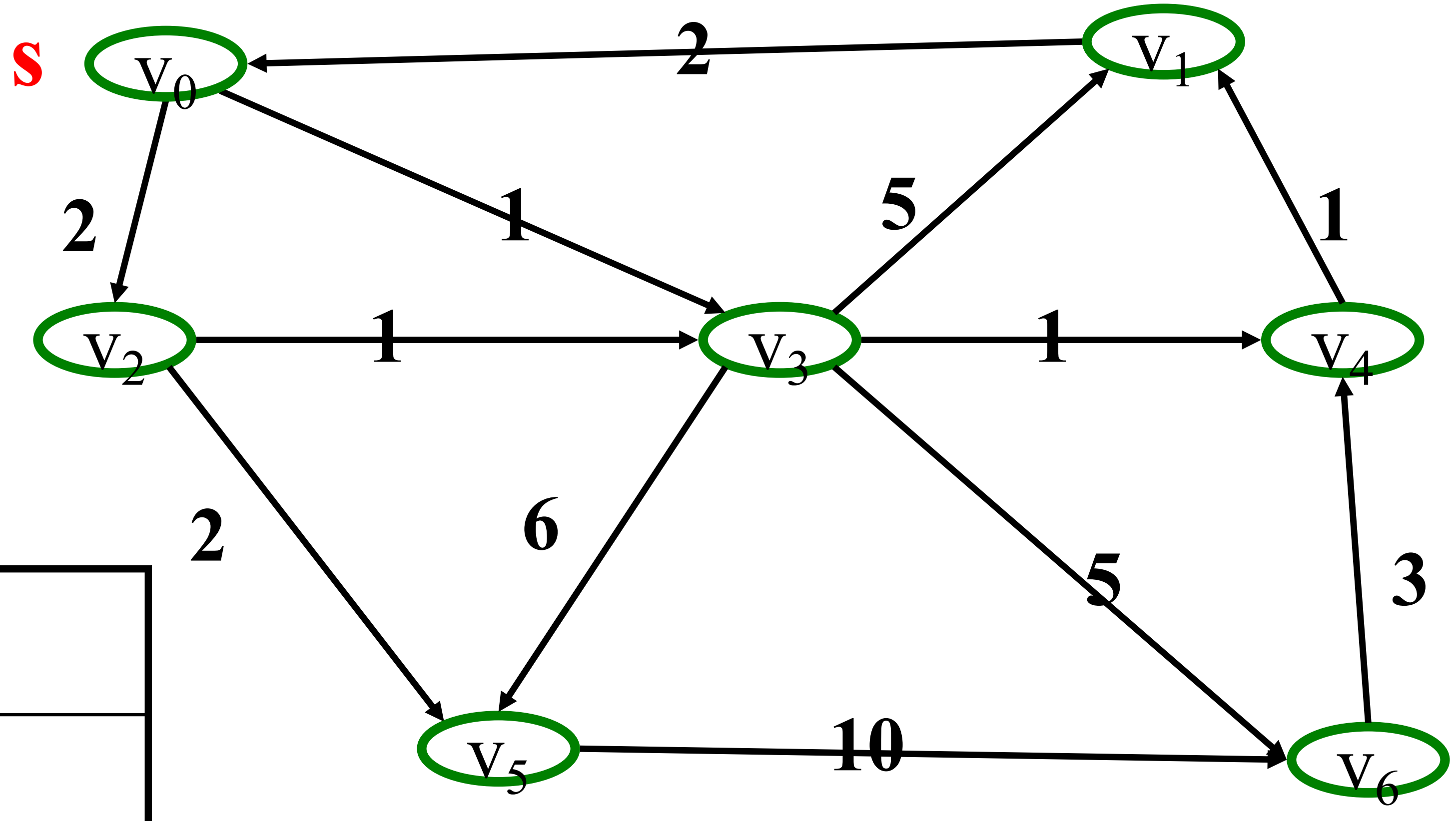
So which is better?

- Sparse: $O(|V|\log|V|+|E|\log|V|)$ (if $|E| > |V|$, then $O(|E|\log|V|)$)
- Dense: $O(|V|^2 + |E|)$, or: $O(|V|^2)$

But, remember these are worst-case and asymptotic

- Priority queue might have slightly worse constant factors
- On the other hand, for "normal graphs", we might call **`decreaseKey`** rarely (or not percolate far), making $|E|\log|V|$ more like $|E|$

Find the shortest path to each vertex from v_0



V	Known	Dist from s	Path
v0			
v1			
v2			
v3			
v4			
v5			
v6			

Order declared Known: